Paper Number: P1068R0

Title: Vector API for random number generation

Authors: Ilya Burylov <ilya.burylov@intel.com>
Alexey Kukanov <Alexey.Kukanov@intel.com>
Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>
Andrey Nikolaev <Andrey.Nikolaev@intel.com>
Pablo Halpern <Pablo.g.halpern@intel.com>

Audience: SG1 (Parallelism & Concurrency)
Date: 2018-05-07

I. Introduction

C++11 introduced a comprehensive mechanism to manage generation of random numbers in the <random> header file.

We propose to introduce an additional API based on iterators in alignment with algorithms definition.

II. Motivation and Scope

The C++11 random-number API is essentially a scalar one. It's stateful nature the scalar definition of underlying algorithms.

However, most existing algorithms for generation of pseudo- or quasi-random numbers allow algorithmic rework to generate numbers in batches, which allows the implementation to utilize SIMD-based HW instruction sets.

Internal measurements show close to linear performance scaling over SIMD-size for most important baseline engines, yielding an order of magnitude performance difference on the table on modern HW architectures.

From user perspective, the proposed new APIs usage is straightforward and aligned with existing practices in the parallel algorithms domain.

Development of the underlying implementation is a different story. Naive implementation of the APIs is trivial (and shown below), but a performance-tuned implementation is an expert-only area.

Vector APIs are common for the area of generation random numbers. Examples:

* Intel(R) Math Kernel Library (Intel® MKL)

  - Statistical Functions component includes Random Number Generators C vector based API

* Java* java.util.Random

  - Has doubles(), ints(), longs() methods to provide a stream of random numbers

* Python* NumPy* library

  - NumPy array has a method to be filled with random numbers

* NVIDIA* cuRAND

  - host API is vector based

Intel MKL can be an example of the existing vectorized implementation for verity of engines and distributions. Existing API is C [1] (and FORTRAN), but the key property which allows enabling vectorization is vector-based interface.

Another example of implementation can be intrinsics for the Short Vector Random Number Generator Library [2], which provides an API on SIMD level and can be considered an example of internal implementation for proposed modifications.

III. Impact On the Standard

This is a library-only extension. It adds new member functions to some classes but does not change any existing functions nor does it add any data members or virtual functions. It should, therefore, be ABI compatible with existing implementations.

IV Summary of changes

The following wording is relative to the C++17 standard. Future revisions of this proposal will include exact sections and deltas.

The engine classes are modified with an additional member function in generating functions section

```
class linear_congruential_engine;
class mersenne_twister_engine;
class subtract_with_carry_engine;
class discard_block_engine;
class independent_bits_engine;
class shuffle_order_engine;
```

Added function (with example of trivial implementation):

```
template<class OutputIt>
void operator()(OutputIt first, OutputIt last) {
    for (; first != last; ++first) {
        *first = operator()();
    }
}
```

Additional iterator version of generate canonical function (with example of trivial implementation):

```
template<class RealType, size_t bits, class URBG, class OutputIt>
void generate_canonical(OutputIt first, OutputIt last, URBG& g) {
    for (; first != last; ++first) {
        *first = generate_canonical<RealType, bits, URBG>(g);
    }
}
```

The distribution classes are modified with two additional member functions in generating functions section

```
class uniform_int_distribution;
class uniform_real_distribution;
class bernoulli_distribution;
class binomial_distribution;
class geometric_distribution;
class negative_binomial_distribution;
class poisson_distribution;
class exponential_distribution;
class gamma_distribution;
class weibull_distribution;
class extreme_value_distribution;
class normal_distribution;
class lognormal_distribution;
class chi_squared_distribution;
class cauchy_distribution;
```

```
    class fisher_f_distribution;
    class student_t_distribution;
    class discrete_distribution;
    class piecewise_constant_distribution;
    class piecewise_linear_distribution;
```

Added functions (with example of trivial implementation):

```
template<class OutputIt, class URBG>
result_type operator()(OutputIt first, OutputIt last, URBG& g) {
    for (; first != last; ++first) {
        *first = operator()(g);
    }
}

template<class OutputIt, class URBG>
result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm) {
    for (; first != last; ++first) {
        *first = operator()(g, parm);
    }
}
```

An optimized implementation should ensure exactly the same result as trivial implementation based on scalar function calls.

Existing library components do not depend on the proposed change, only new APIs added.

V. Design Decisions

Several alternative solutions investigated.

a) Auto-vectorization of scalar interface with compiler.

Most distributions has no explicit issues with vectorization, but they are built on top of compiler possibility to vectorize generate_canonical.

generate_canonical() wrapper function complicates vectorization picture for compiler, but is not a show stopper for vectorization.

The key issues are in underlying engines:

- mersenne_twister_engine<...>::operator()()
- linear_congruential_engine<…>::operator()()
- subtract_with_carry_engine<…>::operator()()

These functions are not vectorizable without human algorithm modification for batch processing, which requires API changes. The main reason - algorithm data dependencies between successive operator calls which compiler cannot reasonably handle.

b) Bufferization of engine results.

There is a possible technical solution, when engine generates a batch of numbers at first call (which can be internally vectorized), returns the first of them and stores others in internal buffer. On further calls, it pops number from the buffer and generates the further batch if buffer becomes empty.

Cons of this approach:

- Instable performance of operator() (requirement of amortized constant complexity is fulfilled though).
- Final solution performance is suboptimal, because we now cannot inline both engine and distribution call at once and keep intermediate results in registers.
- Overcomplicates internal implementation.

c) std::generate + specialization

We can't specialize std::generate with specific random generator, because there is no single "right" way to call std::generate. All alternatives require a user lambda or bind operation:

```
std::default_random_engine eng;
std::normal_distribution<double> dist(0.,1.);

auto my_rand = std::bind(dist,eng);
std::generate(data.begin(), data.end(), my_rand);
```

or

```
std::generate(data.begin(), data.end(), [&]() {return dist(eng);});
```

As such, we can't implement generic enough solution based on specialization.

d) Using `unseq` and/or `vec` execution policy as an additional explicit function argument

Requirement to exactly match the sequence generated by vector API with sequence generated by scalar API makes additional execution policy parameter redundant.

Further investigation should be done to explore opportunities to apply threading in random number generations, which are out of scope of the current proposal.

VI. Proposed Wording

Additional line to be added to requirements tables in requirements for:

* Uniform random bit generator

Expression: g(first, last)

Return Type: void

Effects: stores results equivalent to sequential calls of g() in dereference of every iterator in the range [first, last)

Complexity: no worse than the complexity of n consecutive calls g()

* Random number engine requirements

Expression: e(first, last)

Return Type: void

Effects: stores results equivalent to sequential calls of e() in dereference of every iterator in the range [first, last)

Complexity: no worse than the complexity of n consecutive calls e()

* Random number distriution requirement

Expression: d(first, last, g)

Return Type: void

Effects: stores results equivalent to sequential calls of d(g) in dereference of every iterator in the range [first, last)

Complexity: no worse than the complexity of n consecutive calls d(g)

Expression: d(first, last, g, p)

Return Type: void

Effects: stores results equivalent to sequential calls of d(g, p) in dereference of every iterator in the range [first, last)

Complexity: no worse than the complexity of n consecutive calls d(g, p)

Modification to Header `<random>` synopsis.

The engine classes are modified with an additional member function in generating functions section

```
     template<class UIntType, UIntType a, UintType c, UintType m>
     class linear_congruential_engine {
     ...
       template<class OutputIt>
       void operator()(OutputIt first, OutputIt last);
     };

   template<class UIntType, size_t w, size_t n, size_t m, size_t r,
            UIntType a, size_t u, UIntType d, size_t s,
            UIntType b, size_t t,
            UIntType c, size_t l, UIntType f>
     class mersenne_twister_engine {
     ...
       template<class OutputIt>
       void operator()(OutputIt first, OutputIt last);
     };

   template<class UIntType, size_t w, size_t s, size_t r>
     class subtract_with_carry_engine {
     ...
       template<class OutputIt>
       void operator()(OutputIt first, OutputIt last);
     };

   template<class Engine, size_t p, size_t r>
     class discard_block_engine {
     ...
       template<class OutputIt>
       void operator()(OutputIt first, OutputIt last);
     };

   template<class Engine, size_t w, class UIntType>
     class independent_bits_engine {
```

```
    ...
      template<class OutputIt>
      void operator()(OutputIt first, OutputIt last);
    };

  template<class Engine, size_t k>
    class shuffle_order_engine {
    ...
      template<class OutputIt>
      void operator()(OutputIt first, OutputIt last);
    };
```

Additional standalone function

```
  template<class RealType, size_t bits, class URBG, class OutputIt>
    void generate_canonical(OutputIt first, OutputIt last, URBG& g);
```

The distribution classes are modified with two additional member functions in generating functions section

```
  template<class IntType = int>
    class uniform_int_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class uniform_real_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  class bernoulli_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class IntType = int>
    class binomial_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class IntType = int>
    class geometric_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class IntType = int>
    class negative_binomial_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class IntType = int>
    class poisson_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class exponential_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class gamma_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class weibull_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class extreme_value_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class normal_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class lognormal_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };

  template<class RealType = double>
    class chi_squared_distribution {
    ...
      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g);

      template<class OutputIt, class URBG>
      result_type operator()(OutputIt first, OutputIt last, URBG& g, const
  param_type& parm);
    };
```

```
template<class RealType = double>
  class cauchy_distribution {
  ...
    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g);

    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm);
  };

template<class RealType = double>
  class fisher_f_distribution {
  ...
    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g);

    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm);
  };

template<class RealType = double>
  class student_t_distribution {
  ...
    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g);

    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm);
  };

template<class IntType = int>
  class discrete_distribution {
  ...
    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g);

    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm);
  };

template<class RealType = double>
  class piecewise_constant_distribution {
  ...
    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g);

    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm);
  };

template<class RealType = double>
  class piecewise_linear_distribution {
  ...
    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g);

    template<class OutputIt, class URBG>
    result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm);
  };
```

VII. References

1. Intel MKL documentation:
   https://software.intel.com/en-us/mkl-developer-reference-c-2019-beta-basic-generators
2. Intrinsics for the Short Vector Random Number Generator Library
   https://software.intel.com/en-us/node/694866